Automatic Generation of Context-Sensitive Textual Help

by

Johannes J. de Graaff, Piyawadee "Noi" Sukaviriya, and Charles A.P.G. van der Mast

GIT-GVU-93-11
April 1993

Graphics, Visualization & Usability Center

Georgia Institute of Technology
Atlanta GA 30332-0280
AUTOMATIC GENERATION OF CONTEXT-SENSITIVE TEXTUAL HELP

Johannes J. de Graaff, Piyawadee "Noi" Sukaviriya, and Charles A.P.G. van der Mast
Delft University of Technology and Georgia Institute of Technology

e-mail: J.J.deGraaff@is.twi.tudelft.nl, noi@cc.gatech.edu
telephone: +31-15-785811, +1-404-894-9105

ABSTRACT
Graphical user interfaces tend to get more and more complex, and consequently, the user needs help to work with these interfaces. Development of good help systems is very time-consuming, if only because the help system needs to change whenever the specifications do. Therefor, we have been concentrating on the automatic generation of help.

The context of the research reported in this paper is the User Interface Design Environment (UIDE) [1], which uses high-level application and interface representations to generate interfaces and help. Based on descriptions of an interface, help can be generated to answer two questions: "Why is an object not usable (grayed out)?", and "How can it be made usable?". Pre- and post-conditions, part of the semantics and sequencing control representations in UIDE, are used to answer these questions.

The help system uses unsatisfied preconditions to explain why an object in the interface is disabled. A standard set of templates is used to translate often occurring predicates into text. A planner is used to derive how to enable the object in question. The help system looks at those predicates that must become true to enable the object, and searches for objects with corresponding post-conditions.

KEYWORDS: Context-sensitive help, Automatic help, Textual help, UIDE, User Interface representations.

INTRODUCTION
Modern graphical interfaces tend to get more and more complex, and although users are helped by these interfaces in most ways they are also intimidated by the amount of choices presented. To aid the user in working with these interfaces, help systems are very essential. Creating a proper help system to match graphical interfaces requires a tremendous amount of work, and has high cost of maintenance to keep up with system changes. Yet, the resulting help systems are not context-sensitive and do not correspondingly explain the system dynamics such as why a button is currently disabled.

Shneidermann describes three different kinds of knowledge every user has while working with a computer interface: knowledge about the application task at hand, knowledge about the concepts of a computer, and knowledge about the current dialogue syntax [6]. Our help system focuses on helping users understand the conditions of tasks at hand, which fits in with the first category of knowledge. The other two types of knowledge are also necessary, but are not generated by our help system.

The research effort on help reported in this paper focuses on automatically generating textual help. The automatic generation depends on semantic knowledge representations internally attached to interface objects on the screen. The automatic generation mechanism makes it possible to provide context-sensitive help for an application without much effort. Another advantage of generating the help automatically is that the help system will always be consistent with the application, and changes in the application will automatically result in changes in the help system.

Questions
To help the user with semantic tasks the help system currently answers two questions:

Why is this widget disabled? This question can be asked when a widget is disabled, and the user is not sure about the reasons. The user asks this question by pointing at the disabled object and pressing the HELP key. In this case a short explanation will be presented to the user.
How can this widget be enabled? This question can be asked when the user knows that a certain widget is to be used, but not what to do to enable the widget.

We currently answer these two questions related to standard interface objects. These objects are often called widgets. Examples of widgets are buttons, sliders and textfields. We don’t claim that these two questions will be sufficient to help the user with all problems regarding semantic tasks. They compliment other forms of help, and provide very context-sensitive answers to often asked questions.

Answers

Animated help has already been developed by Sukaviriya [7], and is more suitable for explaining syntactical components of the interface. Text is more suitable for explaining abstract concepts, which is appropriate for explaining application task semantics.

In order to generate textual help for the questions above, we need to have some form of internal representations. Without representations, one could possibly enter explanations directly into the help system. However, the designer could not possibly anticipate all combinations of conditions which would render all interface objects disabled. We take an application-independent approach of representing information such that the help generation mechanism could generate context-sensitive information, yet can work with other applications as well.

First we analyze what kind of information is needed to answer the questions mentioned above.

Why?

In order to explain why a certain widget is disabled, we need to know why the system decided to disable this widget. This information should not be embedded in the code if we want to generate application-independent help. Rather we would have this information easily accessible. One approach is to attach pre-conditions to widgets. These pre-conditions describe which predicates have to be true for the widget to be enabled.

How?

In order to explain how a disabled widget can be enabled we first look at the pre-conditions of the widget. To enable the widget the false pre-conditions will have to become true. This means that we have to know what each widget does, so we can find a widget that has the desired effect on the pre-condition. This information is attached to the widgets using post-conditions. Post-conditions describe the effects of the widget’s invocation on the application.

PRE- AND POST-CONDITION SYSTEM

In the previous paragraphs we have seen a need for pre- and post-conditions attached to interface objects. Our help system extends upon research by Daniel Gieskens [2], who has developed such a system. In order to understand the help generation, we need to give a brief overview of this system.

The key of Gieskens’ system is to describe the dialogue in an interface using pre- and post-conditions. He has augmented widgets with pre-conditions to determine their visibility and enabled/disabled state, and post-conditions that are asserted when certain actions are performed on the widget.

Architecture

The architecture of the system is shown in Figure 1. The three center elements are most important. The current state is the memory of the system. It maintains a number of predicates, and each predicate which is part of the current state is by definition true. The current state is maintained by the predicate manager.

The predicate manager is the only process that can access the current state, thereby ensuring consistency. The predicate manager has several tasks, such as evaluating truth-values for predicates and acting upon post-conditions with requests for removal from or addition to the current state.

The widget manager manages the widgets, and registers their pre- and post-conditions. It also notifies widgets if one of the predicates in their pre-condition has changed, giving them a chance to re-evaluate their state.

![Figure 1: Architecture of the system](image)

Since widgets are always accessed through the widget manager their pre- and post-conditions will always be evaluated by the widget manager. This setup allows the user to specify the complete dialogue of the interface by appending the predicates to the widgets through the widget manager. No application code is required to describe the dialogue of the interface.

Example

The example application we will use throughout this paper is deceptively simple: a cannon. However, despite its simplicity it still contains enough dependencies to allow us to demonstrate the full range of capabilities of the help system. The functionality of the cannon consists of loading,
firing, and switching the safety on or off. Figure 2 shows an interface for this application.

```
[Image of interface]
```

```
del: status(safety.off)        del: status(safety.on)
add: status(safety.on)         add: status(safety.off)

pre: status(cannon, empty)    pre: status(safety.off)
del: status(cannon, empty)    del: status(cannon, loaded)
add: status(cannon, loaded)   add: status(cannon, empty)
```

**Figure 2: Cannon application**

In the pre- and post-condition system, pre- and post-conditions are attached to all interface objects in order to specify the dialogues that can take place in the interface. The pre- and post-conditions are also shown in Figure 2. `pre:` denotes the preconditions for the interface object. For instance, the load button is only enabled when `status(cannon, empty)` equals true. `add:` and `del:` indicate which predicates will be deleted from and added to the current state. For instance, loading the cannon causes `status(cannon, empty)` to disappear from the current state, while `status(cannon, loaded)` appears. The example above, including its pre- and post-conditions, will be used throughout this article.

**EXPLAINING WHY**

We have seen that pre- and post-conditions can be used to determine why a widget is disabled, and how it can be enabled. However, presenting the raw pre-conditions to the user when explaining why a widget is disabled will not help much. Therefore, the predicates have to be translated to text.

**Templates**

The main strategy in converting pre-conditions into text is to use templates. Each predicate which can be used in the system has a template associated with it. This template is used to translate that particular kind of predicate into text. The template contains the English sentence that will explain the meaning of this kind of predicate. The predicate itself will be used to fill in the details of this particular predicate. The idea is to define a standard set of predicates and associated templates which will cover 95% of all cases, and allow for user customization to cover the other 5%.

To demonstrate how templates are used to generate textual help, let's look at an example. We will look at the `status()` predicate. This kind of predicate represents the state of an object. It has two arguments, `object` and `state`. For example, a predicate `status(cannon, loaded)`, in which there is an object, `cannon`, and its state, `loaded`, which altogether says that the cannon is loaded.

The template associated with this kind of predicate will have to explain this kind of predicate. An English sentence to explain this predicate is: "the `<object>` is `<state>`". In this sentence, `<object>` and `<state>` are the details that have to be filled in using a particular instance of the predicate. For example, a widget with precondition `status(cannon, loaded)` is only enabled if the cannon is loaded. If the cannon is not loaded, the help text will be: "the cannon is not loaded".

To produce better text we also allow text generated from templates to be touched up. For instance, in the example above it will make more sense to explain what the state is, instead of explaining what it is not. That is, the help would be better if it said: "the cannon is empty". Obviously, `empty` cannot be derived directly from `not loaded`. A lookup has to be performed to see what the other state of the object is. This alternative state can then be used in the predicate. Note that this particular approach only works when there are only two states in which the object can be.

**Generation**

The routine that generates the help to explain why a widget is disabled is rather straight-forward. The generation is based on master templates of predicates. The slots in templates are filled either by getting information from the object on which help is requested, or by executing functions that generate supplemental text. Figure 3 shows a general structure of the template used to generate the text for "why" explanations. Each box represents the part of text which will be dynamically generated by application-independent functions written for each particular part of the sentence. The functions to determine the name and type of the widget are very simple; they only look up the information in the object. The Reasons are where the real generation takes place. The Reasons function is expanded as shown.

```
[Diagram of architecture]
```

**Figure 3: Architecture of explaining why**

The reasons are generated by Explain Predicates. This function looks at each predicate in turn, and if the predicate is currently false it will be explained. Explain Predicates calls the appropriate function to generate text for the predicate. Each template has its own function, and there is also a generic function to handle all other predicates.

Another example of a template is the template associated with the equals predicate. This predicate evaluates to true when both its arguments are equal. The template for this predicate is: “arguments [doesn't] equal argument2.” Depending on the use of the predicate the doesn't part can either be displayed or suppressed.

Example
In the examples below we do not show original screen-dumps because the readability of such dumps is not very good. We will show a sample screen-dump at the end of the article in Figure 9. Initially the cannon is not loaded, and the safety is on, which is why the fire button is disabled. If the user requests help on the fire button in this situation the help system explains both these reasons, as in Figure 4.

![Figure 4: Explaining why](image)

Now, if the cannon is loaded by clicking on the load button, then the help message changes. It only mentions the safety, which is still on, as in Figure 5.

![Figure 5: Explaining why](image)

EXPLAINING HOW
To explain how a widget can be enabled, we need to derive sequences of actions or “plans”. Unfortunately, planning is not an easy task. Since our research’s focus is not on AI planning, the help system reported here only uses a relatively simple planning algorithm called STRIPS [5]. This algorithm performs well enough to demonstrate all the features of the current help system. It would not hold up in the real world, due to several shortcomings in its structure.

Planner
STRIPS is a simple planner by Nilsson [5]. It makes use of two concepts that are still widely used by planning algorithms. It uses the means-end analysis first used by GPS. With this technique, problems are solved by applying an operator to a goal, and listing the pre-conditions of that operator as new goals. STRIPS itself introduced the action-model, in which each step has post-conditions denoting the changes by that step in the current context. These two techniques are still used by most planners.

To be able to use means-end analysis and the action-model, both pre- and post-conditions must be present. The system underlying the help system attaches pre- and post-conditions to widgets. Our help system takes advantage of this by using these pre- and post-conditions to do the planning. This means that the action representation associated with the widget is represented within the widget itself. An advantage of this model is that no additional knowledge model is needed, apart from the widgets themselves. A drawback is that the representation is rather inflexible, for instance, it only allows for one action to be associated with each widget.

Algorithm
STRIPS is a recursive planner. It is initially called with a list of goals and a copy of the current state. It then tries to evaluate the top goal. If the top goal is also available in the current state, then the goal is removed from the stack, and the associated post-conditions are evaluated.

If the top goal is not in the current state, then the planner searches for an object with post-conditions that could render the top goal true. If it finds such an object, it calls STRIPS recursively to first solve constraints, if any, for the object.

When the top goal is a rule, this means that all its preconditions have been satisfied. This in turn means that the rule is applicable, and its post-conditions will be applied to the current state description and removed from the stack.

Templates
The routine that generates text for how to enable a widget is rather simple, apart from the planning algorithm described above. The architecture is shown in Figure 6. Only one fixed template is needed to generate the explanation. The only variable part consists of the list of steps to be taken. This is just a list of interactions with the interface. This list is generated using the planning algorithm.

![Figure 6: Architecture of explaining how](image)

After all the steps have been derived, they are presented in a sequential order to the user. All the information displayed here can be derived readily from the plan that results from the planning process. Each step is presented with the action that has to be performed on the widget, the widget’s type, and its name.
Example
If we start again with the initial setup, that is, the cannon is empty, and the safety is off, then the help system will explain how to enable the fire button as in Figure 7.

The button "fire" can be enabled by the following method:
- click button "load"
- change safety to off

Figure 7: Explaining how

If we click on the load button, as suggested by the help system, a more complex situation arises. Suppose we want to know how we can enable the load button. Clearly the load button is disabled because the cannon is already loaded. However, to make the cannon empty the cannon has to be fired, but the fire button is also disabled. The correct way to handle this is to first turn the safety off, after which the cannon can be fired. Using the planner the help system also comes to this conclusion, as shown in Figure 8.

The button "load" can be enabled by the following method:
- click button "fire"
- change safety to off

Figure 8: Explaining how

IMPLEMENTATION
The help system has been implemented using C++ and the XView interface toolkit. The use of the XView toolkit caused some constraints with respect to implementation. Originally we had the idea to integrate our help system seamlessly with the default OpenLook help facilities. This proved to be a problem which could only be solved by low-level hacking in the toolkit, something we decided not to do. An example of the current implementation is shown in Figure 9.

Figure 9: Screendump of a help window

The current implementation waits for XView help events, and intercepts these events before the XView toolkit itself can generate help. The help system then identifies the object associated with the event, if any, and generates the help for it by calling the text-generation routines described earlier. Figure 10 gives an overview of this architecture.

Figure 10: Architecture of the help system

FUTURE WORK AND CONCLUSIONS
The current help system has shown that it is possible to generate textual help from specifications. Even in its current state we feel that a lot can be gained from such an approach. One possible domain of applications that could benefit from such a help system is the domain of the consumer electronics and related equipment such as copiers and telephones. More and more of these devices have a small text window that can be used to explain the features and dependencies of the interface.

However, much work still needs to be done. There are several parts of the help system that can be improved upon. The current support for design and implementation, for instance, is still very crude, and needs improvement. The planning algorithm could also be enhanced to be able to cope with more complicated dependencies between widgets. This would require more knowledge about these dependencies, and about the relative importance of widgets in user tasks. The knowledge base would need enhancements in order to maintain this information.

We also realize that text alone does not offer enough help. Therefore, we are currently researching the possibilities of merging textual and animated explanations into a more complete help system [8]. One last interesting option is making the help system user aware. Using user-specific information, the help system could generate help based not only on the context of the application, but also based on the user, e.g. suggesting the users most-used functionality first, or not explaining more advanced functionality to novice users.

Better support for design and specification
The current support for design and specification is still very crude, and only the initiated are able to specify a dialogue with it. The current support consists of an extension to DevGuide, the interface builder from Sun. We have extended DevGuide with the ability to enter the pre- and post-conditions for each interface element. This works fine from a technical point of view. However, it is very easy to loose track of all the dependencies and predicates that have been entered.
One solution to this problem is to use a specification editor such as D2M2, the Delft Direct Manipulation Manager [4]. This specification editor allows for specifying an interface in a consistent and clear manner. One of the benefits of D2M2 is that most information is shown graphically, with diagrams and other graphical representations. In fact, D2M2 already contains a graphical means to describe the dialogue, which could be extended to describe the pre- and post-conditions as well.

**Improved planner**

STRIPS, the currently implemented planner, can be improved upon in several ways. STRIPS doesn’t take certain dependencies between actions into full account, which may cause the planner to fail in case of several related actions with interdependencies. Currently, implementation of a better planner has already started.

Another improvement is to make the planner use hierarchical data. This will address the problem of scale, amongst other things. Instead of seeing each pre-condition as equally important, some pre-conditions can be more important than others, for instance because they denote the start of a higher-level operation. Using this hierarchy the planner can first create a rough high-level plan, and then refine it using the lower levels in the hierarchy. A future extension to provide the hierarchical data is described below.

**Extended knowledge base**

The knowledge base used to describe the pre- and post-conditions is rather terse. Within the framework of UIDE, the User Interface Development Environment, more work has been done to extend this knowledge base [2]. Currently, this knowledge base is being revised. One of the more interesting revision is the introduction of an action hierarchy [8]. This action hierarchy describes the actions and parameters of an application, how these actions and parameters are tied to an interface, and how this interface will behave. This information is structured in a hierarchy, shown in Figure 11.

![Figure 11: UIDE action hierarchy](image)

There are three different levels of action descriptions. The highest level contains semantic actions meaningful to the application. The next level consists of actions that get values or selections from the interface to the application. This level functions as an agent between the two other levels. The last level deals with user interactions with the interface.

**ACKNOWLEDGEMENTS**

We would like to thank Jim Foley for his comments on our work, and for enthusing us. We also want to thank all the students at the Graphics, Visualization and Usability lab for their feedback on our research.

Funding for this research was provided by Sun Microsystems’ collaborative research program and by Siemens AG.

**REFERENCES**


